Rapport de projet

Grand Theft pedestrian

Faveraux Adrien – Lechalier Jc

Sommaire

1. Analyse des besoins

a. Besoins lié au jeu

Nous devons faire un jeu qui s'apparente à un jeu de type chasseur-voleur dans une ville. Dans cette ville il y a des routes, des bâtiments qui sont de différents types et enfin des acteurs qui sont aussi de différents types. Les éléments bâtiments peuvent être de type banque, prison, repère, ou un simple mur et les acteurs sont quant à eux soit des personnages (piétons ou voleurs) soit des véhicules (fourgons blindés, voitures de patrouilles, hélicoptères).
Lorsqu'un piéton est près d'une banque ou d'un fourgon blindé, alors il peut la ou le cambrioler et prendre une certaine somme d'argent, lorsqu'il cambriole alors il est bloquer pendant un certain temps. Il devient alors un cambrioleur. Le but des joueurs des alors d'essayer de l'attraper avant qu'il rejoigne le repère le plus proche.
Pour cela il y a deux méthodes soit on envoie un voiture de police sur la même case que le cambrioleur soit on pose un hélicoptère sur la banque qui se fait cambrioler. Cela a pour conséquence d'envoyer le cambrioleur à la prison la plus proche.
Le joueur doit à tout moment voir la ville sur laquelle il joue, car il doit empêcher que la ville perde de l'argent à cause de vol.

b. Contraintes techniques

Nous devons faire un jeu qui sera multi-joueur, mais aussi le jeu doit être un jeu en réseau. Chaque joueur est sur un ordinateur différent de même que le serveur de jeu.
Ainsi chaque joueur est sur son propre ordinateur et communique via le réseau pour faire les différentes interactions qu'il va bien pouvoir faire avec le jeu. Le joueur devra pouvoir se connecter à un serveur de connexion au jeu.

Le jeu est composé de trois grandes parties. Il y a dans un premier temps la partie serveur de jeu. Cette partie s'occupe de toutes les parties chargements de données en mémoire, distribution des données au Connecteur et gère les figurants.
Ensuite il y a la partie serveur de connexion. C'est à ce serveur que le joueur va tenter de se connecter. Cette partie s'occupe de gérer les requêtes liées aux joueurs. Le rôle principal de cette partie est donc de faire transiter de manière ordonnée les différentes actions des joueurs et des figurants lors d'une partie.
Pour finir nous avons la partie client. Il s'agit en fait du joueur réel. C'est avec cette partie que nous voyons l'interface du jeu, son déroulement en temps réel, ainsi que différentes instructions sur le jeu. Le client se connecte et communique avec le serveur de jeu via le serveur de connexion.
Dans ce jeu, nous devons utiliser des techniques de programmations système pour que le jeu consomme le moins de ressources possible en optimisant les temps de calculs des différents processus que nous allons créer ainsi que les requêtes que nous allons faire sur le réseau.

2. Organisation des outils

Le serveur est le programme principal du jeu puisque c'est là que nous allons stocker toutes les informations relatives au jeu. Cette partie du jeu sert à charger les données en mémoire, les gérer ainsi que veiller à l'intégrité des données du jeu de plus il gère aussi le déplacement de chaque figurants sur le terrain. L'intégrité du jeu est gérée de plusieurs façons qui seront décrite plus en détail dans la suite du rapport.

a. Représentation des données

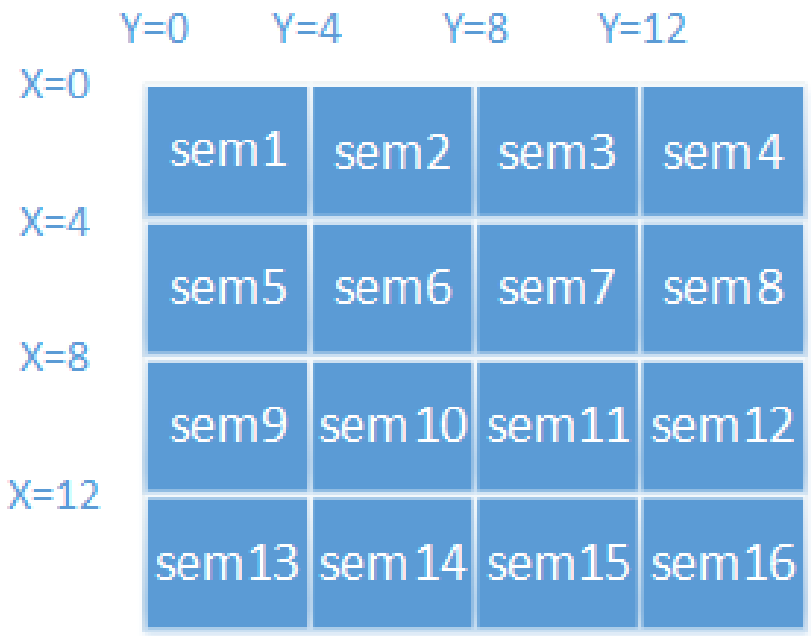
Pour que le jeu soit fonctionnel, il nous a fallu faire en sorte de stocker les données utiles dans la mémoire du système et les partager entre chaque processus sauf le client. Nous avons dû utiliser un outil ipc pour cela. La mémoire du jeu devait être dans un segment de mémoire partagée. Cette mémoire est accessible par l'ensemble des processus d'une machine pourvu que nous ayons la clé du segment (adresse du segment mémoire) et que nous y soyons attachés (accessible en lecture ou lecture/écriture avec la fonction shmat).
Pour que ce l'accès à cette mémoire partagé soit simple nous devons faire en sorte que ce qu'elle contient soit bien représenté.

b. Accès au segment mémoire

L'accès aux données du segment mémoire étant « libre » et que par conséquent sa lecture est possible par plusieurs processus, nous devons contrôler ces accès pour que les informations qui s'y trouve soit vrai en tout temps pour le processus qui souhaite exécuter une action sur le segment mémoire. Pour cela nous disposons d'outils ipc : les sémaphores. Ils permettent de bloquer une partie d'un programme si l'action qu'il souhaite faire peut poser problème dans un environnement réparti ou concurrentiel.

La carte est un tableau de caractères qui renseigne comment la carte est représenté pour l'affichage. Or chaque éléments du jeu (piétons, véhicules, joueurs) la modifie en permanence, avec pour certain le caractère aléatoire de leurs déplacements. C'est pourquoi la modification de cette carte doit se faire en verrouillant les sémaphores qui représentent la carte sur le terrain en fonction de leurs coordonnées. Le terrain est divisé en groupe de 4 cases, chaque sémaphore ne verrouille que 4 cases du terrain.

Lorsque le figurant ou le joueur se trouve à la limite entre deux zone les deux zones sont verrouillées pour éviter qu'un figurant de m'écrase si il vient d'une autre zone ou prenne la case ou je voulais aller.



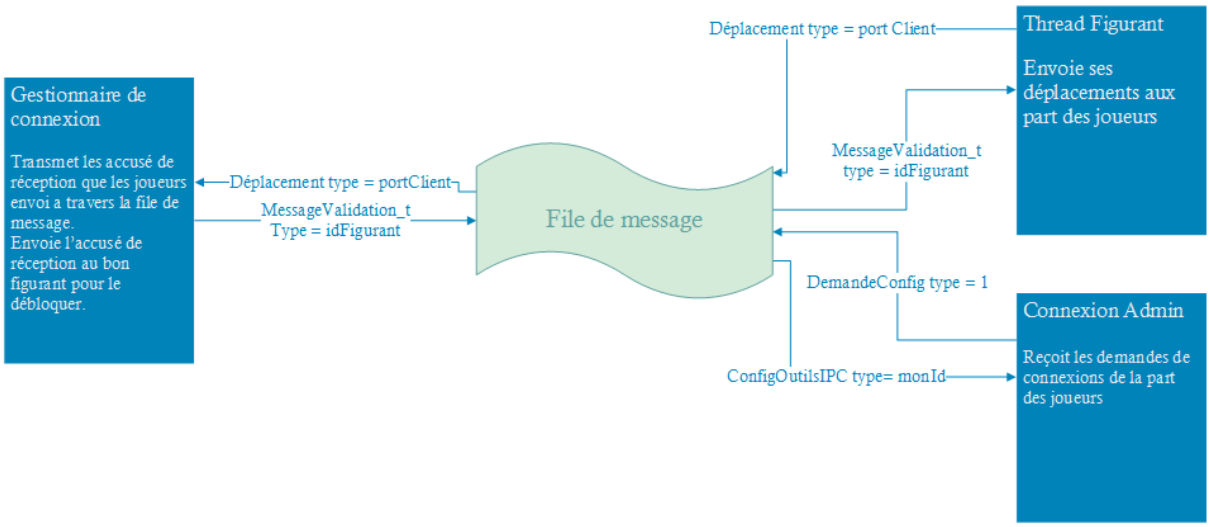
c. File de message

La file de message est, dans notre serveur un outils clé. Il est utilisé pour faire transiter les données entre le serveur de connexion et l'ensemble des threads qui simulent des Figurants dans le jeu.

La file de message est créé au même moment que l'ensemble des outils ipc, lors du démarrage du serveur. Lorsque le serveur de connexion démarre, il va être le premier à écrire dans la file de message. En effet nous récupérons l'ensemble de la configuration mémoire par une requête dans la file de message.

Pour que la file de message soit utilisée de façon optimisé, il faut que chaque message soit identifiés afin qu'il soit lu par son destinataire. De cette façon nous ne risquons pas de saturé la file de message avec des messages jamais lu.
Nous utilisons pour cela le pid du destinataire. Cette méthode est dans un premier temps utilisée par le thread Figurant pour en avertir les joueurs. Les joueurs vont alors à leur tour répondre au Figurant que son déplacement a bien été pris en compte. Pour cela le port client envoie une structure de type MessageValidation à son commanditaire.

Dans un second temps, nous faisons en sorte de ne pas écrire n'importe quand dans la file de message. Chaque message dans la file est une conséquence d'une action généré auparavant. Par exemple le thread Figurant écrit dans la file de message son déplacement pour les deux joueurs. Il va alors attendre leurs réponses et ne rien faire d'autre. En réponse, les ports des clients répondent leur accusé de réception respectif une fois le traitement du déplacement effectué.



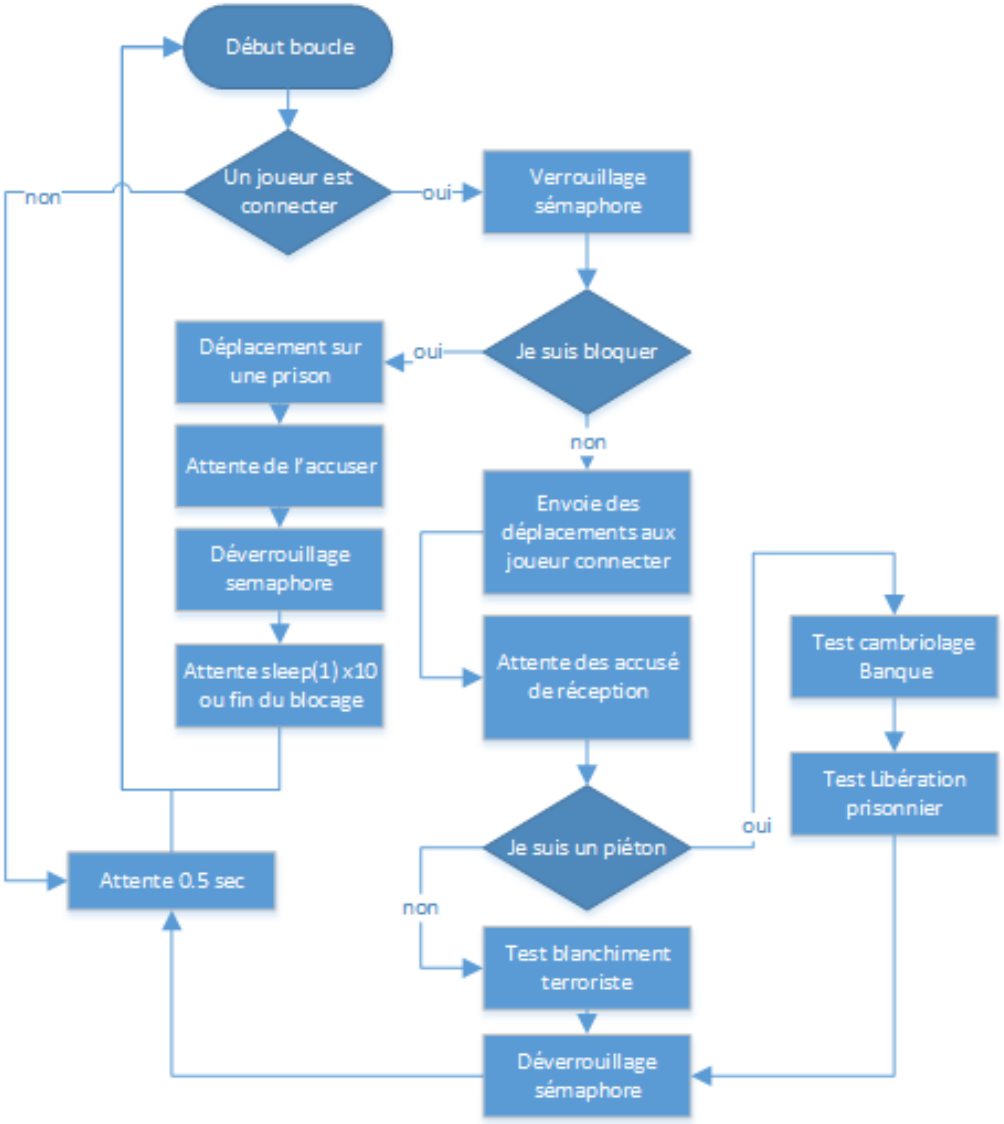
3. Détails des figurants

a. Déplacements des figurants

Dans ce jeu chaque figurant est simulé par un thread. Chaque thread créé un déplacement par rapport à sa position actuel il verrouille sa partie de terrain qui sont représentées en blocs de 4 cases du terrain.
Ensuite il vérifie dans le segment de mémoire si un joueur est connecté au jeu. Si aucun joueur n'est connecté alors il se met en pause et n'effectue aucun déplacement et retentera toutes les 0.4 secondes.
Si au moins un joueur est connecté (son pid a été placé en mémoire partagée), alors on envoie le déplacement que l'on compte faire dans la file de message en direction de tous les joueurs connectés, un message par joueur connecté.
Une fois le déplacement envoyé dans la file de message, on se met en attente d'une réponse (un accusé de réception) de la part du client joueur. Lorsque nous avons l'accusé de réception du client joueur comme quoi le déplacement a bien été effectué alors nous l'écrivons dans la mémoire partagée et nous déverrouillons les sémaphores verrouillés précédemment.

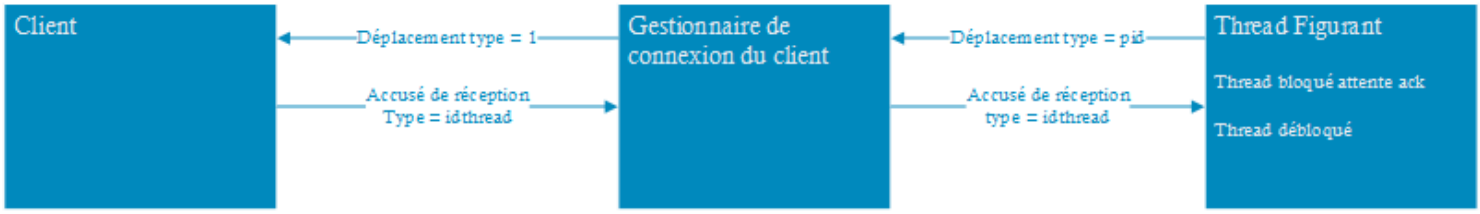
Cela a deux avantages, dans un premier temps cela évite de surcharger la file de message avec des tentatives. Ensuite cela permet de libérer du temps cpu. En effet lorsqu'un processus attend son message il ne fait rien d'autre.

Exemple : un Thread piéton en action



b. Accusés de réception

Les accusés de réceptions ont dans notre application un rôle crucial. Lorsqu'un thread figurant veut se déplacer il verrouille les sémaphores de la zone autour de lui pour que personne d'autre que lui ne puisse accéder à cette zone de la carte.
Il envoie donc son déplacement mais il doit attendre de s'assurer que toute la chaine de processus ai bien été mise à jour leur carte avec le nouveau déplacement (les deux joueurs). Car sinon cela pourrai engendrer des déplacements où deux figurants qui ne devrai pas se marcher dessus, se marche dessus sur la carte des clients.
Donc le thread une fois le déplacement envoyé à tous les joueurs connectés attend de recevoir un accusé de réception avant de débloquer le sémaphore et donc d'autoriser d'autres déplacements dans la zone



c. Fonctionnement d'une prison

Une prison dans notre jeu est juste une case sur la carte. Mais pour en simuler le comportement, à savoir bloquer le piéton qui était devenu cambrioleur et s'est fait prendre, il faut que ce dernier soit bloqué et ne puisse plus pouvoir se déplacer.
Afin de bloquer, stopper un piéton nous utilisons une simple variable (blocage) qui va nous permette de savoir si nous continuons l'exécution de la routine. Chaque tentative de déplacement d'un piéton commence par le test de cette variable. Si celle-ci est égale à 1 alors nous ne pouvons pas nous déplacer et nous entrons dans la procédure de blocage. Dans cette procédure nous recherchons la prison la plus proche sur la carte, on envoie un déplacement à chaque client pour se déplacer dans la prison, à la réception de l'accusé de réception, on change les coordonnées du terroriste sur la carte pour le mettre en prison.

Une fois en prison le client se met dans une boucle, il vérifie toute les secondes si il est toujours bloqué car si un figurant a libéré les prisonniers il sera plus bloqué. Il décrémente aussi un compteur qui sert à savoir combien de temps sa peine va encore durer, par défaut cette valeur est à 10 secondes. Une fois qu'il est libéré on remet son type à Pieton et on le laisse faire ses déplacements normalement en remettant cette variable de blocage à 0.

d. Libération d’un prisonnier

Le système de libération d’un prisonnier est simple, puisque le prisonnier vérifie toutes les secondes si il est encore bloqué, c’est-à-dire qu’il purge encore sa peine, il suffit pour le débloquer de mettre la valeur bloqué à 0. Cela peut être fait directement par les threads figurants Pieton lorsqu’il passe à côté d’une prison.

Pour qu’il soit libéré, un prisonnier à plusieurs méthodes. Soit il a purgé sa peine. C’est-à-dire qu’il a été réveillé 10 fois (il passe donc 10 secondes en prison). Soit il y a un piéton qui passe à côté de la prison (20 % de chance qu’un prisonnier soit libéré). Lorsqu’un piéton passe à côté d’une prison il va alors tenter de faire libérer un prisonnier.
Lorsqu’un cambrioleur est en prison, le thread qui le simulait est endormi et attend que les conditions de sa libération soit remplit. Le thread est endormi avec un sleep() il attend donc la fin du temps pour se réveiller, ce timer est réglé à 1 sec. Une fois qu’il se réveille, il vérifie si les conditions de sa libération sont remplit sinon il se remet à dormir.

e. Cambriolage

Le but du jeu est d’éviter que la ville se soit fait dérober tout son argent qu’elle a, réparti dans des banques et des fourgons. Mais pour perdre de l’argent, il faut qu’il y ait des cambriolages qui soient commit. Dans notre jeu, les Figurants ont deux façons de commettre de cambriolage.

Le premier type de cambriolage est que le piéton cambriole une banque. Pour cela il va après avoir fait son déplacement, envoyé son prochain déplacement au joueur connecté au même emplacement qu’il est actuellement (il reste sur les lieux, c’est plus pratique pour commettre un vol) mais changeant de type (Pieton>Terroriste) avec un butin aléatoire qu’il est en train de voler, il mettra aussi dans la mémoire partagée qu’il est devenu Terroriste. Comme cela le joueur pourra tenter de l’arrêter quand il sera en train de commettre son cambriolage.
Une fois que le terroriste commence son cambriolage, il se met dans une boucle while avec deux conditions pour finir. Cette boucle va tout simplement simuler la longueur du cambriolage. Car oui un cambriolage ça prend un peu de temps à faire.
La première condition de sortie de notre temps de cambriolage est que si le figurant est bloqué (bloque = 1) alors cela veut dire qu’il s’est fait prendre par un client. Dans ce cas il est envoyé de la même manière que si il commençait son cambriolage (c’est-à-dire qu’il change de type) et qu’il était arrêté.
S’il se fait prendre alors l’argent est remis dans le coffre de banque.
La deuxième condition pour finir son cambriolage, est qu’il ait fait 10 tours de boucles car il aura donc mis 10 secondes à cambrioler la banque (belle performance).

La deuxième façon de voler de l’argent dans notre belle ville, consiste tout simplement à voler un fourgon. Mais il ne faut pas qu’on puisse les voler alors qu’ils roulent. Pour cela le thread qui simule l’anciennement Pieton (ou nouveau Cambrioleur, tout est une question de point de vue), va modifier l’état du fourgon.
Mais dans un premier temps, lors d’un déplacement, les sémaphores de la zone sont verrouillés. Ils bloquent donc le déplacement de tout type de figurant mais je ne peux pas le maintenir verrouillé tout le temps du cambriolage sinon mes joueurs ne pourront jamais approcher le terroriste !

Dans son code le fourgon vérifie juste après le verrouillage des sémaphores si il est bloqué comme cela il sera que une fois que le terroriste fera son cambriolage et aura relâchés les sémaphores, le fourgon s’apercevra qu’il est bloqué.

Une fois bloqué le fourgon se réveille toute les secondes pour savoir si il est encore bloqué. Tant que le terroriste n’aura pas remis le token bloque à 0 le fourgon restera bloqué. Comme cela même si on modifie le temps que le terroriste cambriole cela ne posera aucuns problèmes.

A la fin de son cambriolage, le terroriste débloque le fourgon et tout le monde peut repartir.

Pour bien comprendre comment se passe un cambriolage il ne faut pas oublier que dans notre jeu nous avons une variable qui est partagée par tous les Figurants et même les joueurs. Il s’agit ni plus ni moins que du nerf de la guerre ; l’argent. Alors si nous ne protégeons pas cette variable lors des cambriolages, nous risqueront d’avoir des petits problèmes de trésoreries. Pour éviter ce problème, il faut bien la protéger notre variable. Nous utiliserons donc un mutex. Pour pouvoir réaliser un cambriolage en bon et du forme il faudra réussir à prendre le mutex. Cela garantira que le vol sera bien comptabilisé.
Cela est tout à fait et heureusement valable pour la remise du butin.

f. Explication des arguments des threads Figurant

Lors du démarrage d'un thread Figurant, on lui passe la structure d'argument suivante :

```
typedef struct
{
  int i; // position en memoire
  int id;
  enum type_figurant_e type;
  void * memoire_partagee;
} argumentThread_t;
```

Le paramètre i renseigne la position qu'a l'élément dans le tableau de Figurant du segment mémoire.

Le paramètre id renseigne l'id que va avoir le thread. Ce paramètre est utilisé pour savoir a qui envoyer les accusé de réception lorsqu'il va tenter un déplacement.

Le paramètre type est en fait le type qu'a le Figurant à savoir Pieton ou Fourgon.

Le paramètre mémoire_partagee est en fait le pointeur vers le segment mémoire.

Avec ces informations le thread est en mesure de savoir qui il est, comment il doit réagir et aussi où il doit écrire dans le segment mémoire.

4. Serveur de connexion

Le serveur de connexion est accessible par les joueurs qui souhaitent se connecter au jeu Lorsqu'un joueur veut se connecter, il envoie une requête de connexion avec le type de joueur qu'il souhaite incarner dans la socket udp qu'il aura créé avec les paramètres (adresse+ port) qu'on lui aura passé en paramètre au démarrage. A la réception de cette requête, le serveur va alors vérifier dans la mémoire partagée si le joueur peut se loguer (est ce qu'il n'y en a pas déjà un de mon type) dans les deux cases joueur disponible en mémoire. Si il peut alors le serveur lui créé un processus fils avec une socket spécifique sur laquelle le client va pouvoir se connecter pour accéder au jeu.

a. Transmission des messages

Le but premier du serveur de connexion est de transmettre des messages entre le joueur et le serveur de données, la question est comment organiser tout cet engrenage Si les threads se déplaçais seulement dans la mémoire partagée il n'y aurai aucun moyen pour le gestionnaire de connexion de savoir qui s'est déplacé et quand.

On va donc utiliser la file de message pour transmettre des demandes de déplacements des threads figurants jusqu'à chaque ports fils du gestionnaire de connexion. Ces messages étant à l'initiative des Figurants eux-mêmes. Ils savent où envoyer les messages grâce au pid du port joueur. Pour y accéder, rien de plus simple nous avons fait en sorte qu'il soit dans les paramètres de configuration du segment mémoire.

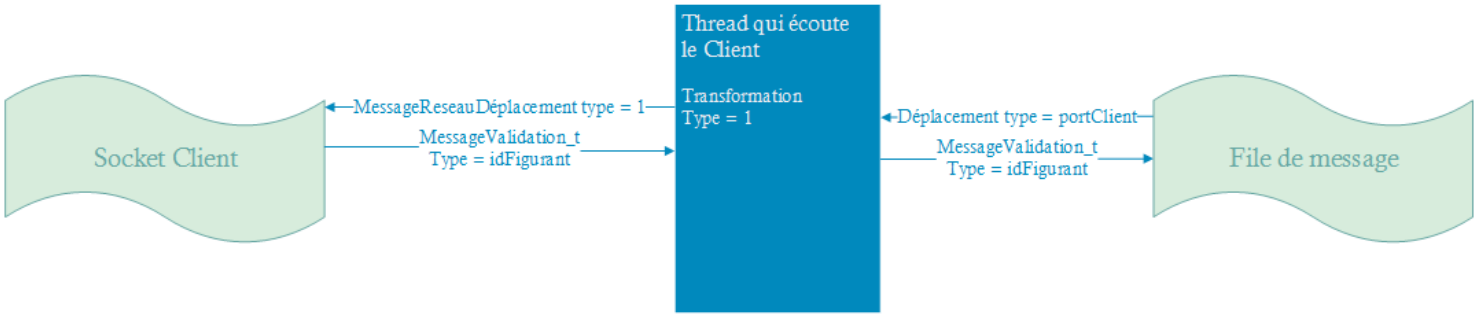
Le gestionnaire va reconnaitre quels paquets lui sont adressés car il y a un seul joueur par port et donc pour que le joueur soit déclaré connecter par le jeu il faut que son pid dans la mémoire partagée soit placé (différent de 0). Les threads figurants envoient donc dans la file de message un message avec le type correspondant au pid du joueur ainsi donc seul le port ou est connecté le joueur recevra le message.

Une fois le message reçu le gestionnaire de connexion change le champ type (type= pidPortClient > type = 1) du message qu'il reçoit, pour que le client reconnaisse un message de déplacements en ensuite il l'écrit directement dans la socket.

Ensuite le gestionnaire de connexion va lire dans la socket le type du message (donc un long)
Il va ensuite appeler une fonction différente en fonction du message reçu, il y a trois types de message que le Gestionnaire de connexion peut recevoir.

Dans un premier temps, il y a les messages accusés de réception Ce sont des messages qui doivent être envoyé au thread Figurant qui a envoyé le déplacement pour qu'il puisse se débloquer. Sans ce message le thread figurant restera bloqué indéfiniment.

Ensuite, il y a les messages de déplacements du client. Ce sont des messages qui sont envoyés du client quand un joueur fait une demande de déplacement à des coordonnées spécifiés. Pour ce type de message le client attend un accusé de réception pour pouvoir afficher le déplacement que le joueur a demandé. Le principe reste le même que si ce message venait d'un thread Figurant.



Enfin il y a les messages d'ordres. Ce sont des messages qui sont envoyés par le client quand le client demande une pause/play ou que le client veut quitter le jeu.
Cette requête ne va pas générer de suite au niveau des messages à proprement parler. Elle les stoppe plutôt en mettant le pid du joueur a 0 en mémoire partager.

b. Représentation d’un déplacement

Dans notre jeu tout le problème de communication réside dans le transport et la pertinence de la carte à travers tous les membres du jeu (serveur, et joueurs). Nous sommes donc obligés d’envoyer cette carte dans le réseau.
Mais il y a plusieurs façons dont nous pouvons structurer nos requêtes afin d’envoyer cette carte du jeu. En effet, pour échanger les données à travers les sockets et la file de message, nous n’avons pas de façon toute faite pour le faire. Nous devons donc proposer et tester différentes méthodes.

La première méthode consiste à de transmettre au client toute la carte toute les 0.5 secondes ce qui permettrai de faire déplacer tous les figurants. Mais cette solution a été rejetée car il faudrait verrouiller toutes les sémaphores à intervalles régulier. De plus la charge réseau serai assez élevée pour ce type de jeu. A savoir que le coup en temps machine est aussi élevé. Car verrouiller des sémaphores est très couteux, alors verrouiller tout un tableau de sémaphores l’est encore plus.

La deuxième méthode serai de transmettre que le carré qui a été verrouillé par le figurant. Nous gagnerons donc en temps processeur car on ne verrouillera qu’une zone.
Cela permettrai de redessiner qu’une partie du terrain, mais cela reste compliqué à mettre en œuvre (remise à la bonne place de la zone en question) et à un coup qui pourrai être réduit en charge réseau.

La troisième méthode, celle que nous avons choisis consiste à envoyer seulement le/les coordonnées sur laquelle/lesquelles les déplacements se font.

Structure Déplacement

La structure de déplacement est donc construite de cette façon.

```
typedef struct {  // type 1 == deplacement
  long type;
  char nom[50];
  enum type_figurant_e figurant;
  int figurantid;
  int argent;
  int joueur; // savoir si le message vient du joueur
  position_t position;
} MessageReseauDeplacement_t;
```

Le champ type est utilisé du côté des threads figurants pour la file de message IPC elle est alors configurée avec le pid du joueur pour pouvoir que le gestionnaire de connexion la récupère.
Elle est aussi utilisée au niveau du gestionnaire de connexion vers le client. Le type = 1 indiquera au client que c'est un message de déplacement qu'il doit lire et traiter en conséquence (affichage et envoie d'un accusé de réception).

Le champ nom n'a de valeur que si c'est un déplacement de figurant. En effet chaque figurant possède son propre nom. Il l'envoie donc en même temps que son déplacement.

Le champ figurant est utilisé pour savoir quel type de figurant se déplace, un Pieton, un Fourgon ou un Terroriste. Ce champ est utilisé pour passer du statut de Pieton à celui de Terroriste et vice versa. Cela nous permet de savoir quand un piéton a décidé de passé à l'acte. Au niveau du client, chaque figurant possède un id propre qui l'identifie. Le client enregistre chaque figurant qui lui envoie un message si elle voit un changement de type par rapport au message de déplacement précédent alors c'est que le Figurant a changé de type.

Nous en parlions plus tôt, le voici, le champ figurantid. Il permet de reconnaitre de manière unique fiable à quel Figurant est associé le déplacement que l'on traite.

Le champ argent est défini quand on passe Terroriste. Il s'agit de la somme d'argent qu'il a dérobé. Ce champ même si il ne paraît pas très utile, permet au client d'avoir facilement accès aux informations sur l'argent. La somme d'argent restante dans la partie sera la différence de l'ancien capital avec cette nouvelle perte. Cela permet aussi de répartir les charges de travails sur le client pour ce qu'il s'agit des calculs de fin de partie.

Le champ joueur est un token qui permet au gestionnaire de connexion de savoir si c'est un déplacement de joueur ou un déplacement de figurant et d'agir en conséquence.

Le champ position est une structure avec deux int x et y qui correspond à la case sur laquelle on veut se déplacer. Ni le client ni le gestionnaire de connexion ne vérifiera qu'il fait un déplacement d'une seule case, on peut donc par exemple aller directement en prison

Structure de réponse à un déplacement

```
typedef struct
{
        long type;
        int figurantid;
        int joueur;
        position_t position;
} MessageValidation_t;
```

Le message de validation est envoyé pour valider un déplacement soit de thread figurant soit de déplacement utilisateur.

Le champ type est utilisé du côté des threads figurants pour la file de message IPC elle est alors configurer avec l'id du figurant pour pouvoir que le thread figurant le récupère.
Le champ type est aussi définit pour valider un déplacement de joueur alors se champ vaudra 2 pour le différencier du message de déplacements.

Le champ figurantid permet de reconnaitre de manière fiable quel déplacement est associé à quel thread figurant.

Le champ joueur est un token qui permet au gestionnaire de connexion de savoir si c'est un déplacement de joueur ou un déplacement de figurant et d'agir en conséquence

Le champ position est la position sur laquelle le joueur s'est déplacé sur la carte pour un thread figurant ou pour un joueur c'est la position sur laquelle il est autorisé à se déplacer

c. Déplacement du joueur

Le gestionnaire de Connexion ne gère pas directement où le joueur va se déplacer mais c'est lui qui va faire la démarche pour qu'il se déplace en toute sécurité sur la mémoire partagée.

Premièrement tout commence par un message de déplacement de joueur sur la socket.
Le gestionnaire de connexion va traiter le message et va essayer de verrouiller le ou les sémaphore(s) du terrain. Cela dépend directement de notre zone de départ celle d'arrivé de notre déplacement. Mais comme on le sait un verrouillage de sémaphore n'est pas réussi à tous les coups.

Si on arrive à verrouiller les sémaphores on regarde en fonction du type de client du joueur si il peut de déplacer en fonction des figurants présent et du décor présent dans la zone. Si le déplacement est impossible alors on abandonne la requête de déplacement.
Alors que si le déplacement est permis, alors nous modifions la carte dans le segment de mémoire partagée. Enfin nous allons vérifier si en fonction du type de client, je peux capturer un terroriste, si je peux, alors je le bloque (bloque=1). Il ira directement en prison lors de sa prochaine tentative de mouvement.

Une fois le déplacement effectué, il fait que l'autre joueur en soit informé. Pour cela le port client qui s'occupe de mon joueur lui envoie un message via la file de message (type = pidClient2). Tout cela est fait si l'autre joueur est bien connecté.
Un problème se pose alors, comment l'autre joueur sera que c'est un joueur et non un figurant qui lui envoie un message.
Heureusement nous avons le paramètre joueur, il trouve tout son sens pour ce cas de figure.
De manière générale, un client se déplace en envoyant un déplacement et reçoit un accusé de réception. Le seul cas possible ou il recevra un message de déplacement avec le token joueur sera quand un autre joueur qui lui enverra directement un déplacement.
Du fait que ce soit un joueur qu'il l'ai envoyé, on ne va pas faire d'accusé de réception. Même si le paquet est perdu au prochain déplacement de joueur il apparaitra à la bonne position.

Puis enfin on envoie un accusé de réception au joueur à l'origine du déplacement pour qu'enfin il le valide et l'affiche sur sa carte.

Mais revenons au fait qu'au tout début nous tentions de verrouiller des sémaphores. Il nous reste un cas ; celui où nous n'arrivons pas à les verrouiller.

Dans ce cas on ne peut pas attendre que les sémaphores se déverrouillent. Cela n'est pas possible car chaque thread figurant attend d'avoir son accusé de réception avant de déverrouiller les sémaphores qu'il a lui aussi verrouillé dans l'attente de validation de son mouvement. Mais si notre gestionnaire de connexion lui aussi est bloqué sur le verrouillage des sémaphores il ne pourra jamais transmettre les accusés de réceptions ou les déplacements de joueur. Pour ces raisons nous sommes potentiellement dans une situation d'inter-blocage.

Pour palier à cette situation, on va enregistrer le déplacement que je joueur veut faire dans une sorte de variable tampon, puis on essaye de verrouiller les sémaphores en NO_WAIT (on y arrive tant mieux sinon on continue).
Si le verrouillage fonctionne, on effectue les mêmes étapes que dans le cas où on aurai réussi sinon on traitre les déplacements des threads, après chaque déplacements de thread traité on retente de déverrouiller les sémaphores. Au bout d'un moment on réussira et on effectuera le déplacement.

d. Gestion des ordres

Le gestionnaire de connexion peut recevoir 2 types d'ordres de la part du client. A savoir la pause/reprise ou l'ordre de quitter le jeu.

Lorsque le gestionnaire de connexion reçoit l'ordre de quitter le jeu, il ne peut pas se quitter sans vérifier si cela n'a pas de conséquence sur le reste le l'application, sans le faire signifier. Pour cela il doit mettre le pid du joueur qu'il gère à 0. Il s'agit d'un paramètre qu'il a rempli en démarrant dans le segment de mémoire partagée. Cela aura pour effet que les threads n'enverront plus de déplacements au joueur.

Lorsque l'on reçoit cet ordre c'est que le client va se déconnecter mais pourquoi ne pas quitter le fils du gestionnaire de connexion directement quand on reçoit cet ordre ?
Pour rappel, les threads Figurant on pour principe d'attendre leur accusé de réception pour valider leur déplacement. Si nous quittons précipitamment le port du client alors qu'un Figurant attend (sémaphores verrouillés) son accusé de réception, nous bloquerons une partie de la carte. Pour pallier à ce problème, il suffit d'attendre que le client traite tout les déplacements et ferme la socket tcp quand il aura finit, pour enfin quitter en toute sécurité.

L'autre ordre est celui de pause/reprise de la partie. Afin de comprendre comment nous avons mis en place cette fonctionnalité, il faut savoir que puisque les threads Figurants sont bloqués en attente d'accusé de réception on va utiliser ce principe pour faire la fonction play/pause (chaque Figurant sera bloqué soit par l'attente d'un accusé de réception soit par l'attente de la libération de la zone).
Pour cela on va mettre une variable de blocage qui quand il est à 1 bloque les transmissions des déplacements au joueur, ils restent donc dans la file IPC et les déplacements des joueurs ne sont pas traité et donc en suspens. Voilà le principe même de la pause.
Cette variable de blocage quand on la change passe soit de 0 à 1 soit de 1 à 0. On a donc bien un principe de mise en pause et redémarrage à la réception d'une requête.

Nous avons donc bloqué un port joueur mais ce n'est pas le cas pour l'autre joueur. Pour cela on a donc implémenté un gestionnaire de signal pour les signaux SIGUSR1 (bloquer) et SIGUSR2 (débloquer).
De cette façon, la communication est donc bloqué/débloqué sur tous les ports clients du gestionnaire de connexion. Donc quand l'un reçoit un message d'ordre l'autre est averti par un signal. Ainsi un des joueurs peut faire pause et l'autre faire reprendre sans le moindre problème.

5. Client : Joueur

Le joueur du jeu doit se connecter au serveur de jeu pour cela nous avons l'adresse et le port d'écoute du serveur pour démarrer notre client. Il se connecte donc au serveur qui nous créer une connexion tcp propre pour notre application.

a. Interface client

Le client peut voir déroulement du jeu grâce à une représentation de la carte en console. Cette interface est surélevée avec ncurses qui permet de prendre en compte la souris et le clavier. De ce fait nous ne sommes pas obligés de rentrer des coordonnées pour déplacer notre hélicoptère ou notre voiture de patrouille.

Cette interface est organisée en trois fenêtres. Il s'agit en fait de bloc d'affichage dans notre fenêtre graphique du joueur.
Dans un premier temps, il y la fenêtre information. C'est ici que vont être notifié l'ensemble des changements d'états des Figurants. Cette partie sera décrite plus en détails dans la suite du rapport.
Ensuite nous avons la fenêtre ville qui quant à elle comporte la carte du jeu. C'est là que l'on voit l'ensemble de ce qui se passe dans la ville (déplacements notamment). C'est aussi dans cet espace que nous allons cliquer pour changer la position de notre voiture de patrouille ou l'hélicoptère en fonction du type de joueur que nous incarnons.
Enfin il y a la fenêtre légende. Il s'agit de la légende avec laquelle nous allons pouvoir identifier rapidement quelle type d'éléments j'ai sur ma carte.

b. Démarrage d'un Joueur

Lorsque le client est créé la première chose qu'il fait est d'envoyer une demande de connexion en udp sur le parent des gestionnaires de connexion (processus père). Il va vérifier si un client du même type n'existe pas déjà. S'il en existe déjà le client se quitte en erreur.

S'il y a de la place le parent des gestionnaires de connexion crée un fils et renvoie le port sur lequel on pourra se connecter.

Le client, suite à cela va s'y connecter, ensuite vient vraiment la phase d'initialisation, le client envoi son type (Voiture ou Helico) au gestionnaire de connexion (fils qui va s'occupé de lui), puis va lire sa position de départ que le gestionnaire de connexion va lui attribuer aléatoirement. Ensuite le client va lire la somme d'argent total de la partie qu'il rejoint, si cette somme arrive à 0 alors le client se quitte ; la partie est perdu.

Ensuite il va charger 2 int qui seront la taille x et y de la carte, on va allouer l'espace nécessaire pour pourvoir la stocker.

Ensuite on va lire les messages réseaux pour récupérer la carte ils peuvent éventuellement être en plusieurs trames de 500 caractères chacune. Cela est fait lorsque la carte du jeu est très grande.

Une fois la carte chargée on crée l'interface graphique pour le jeu.

Une fois ceci fait il ne reste plus qu'a lancer les threads et l'initialisation est terminée.

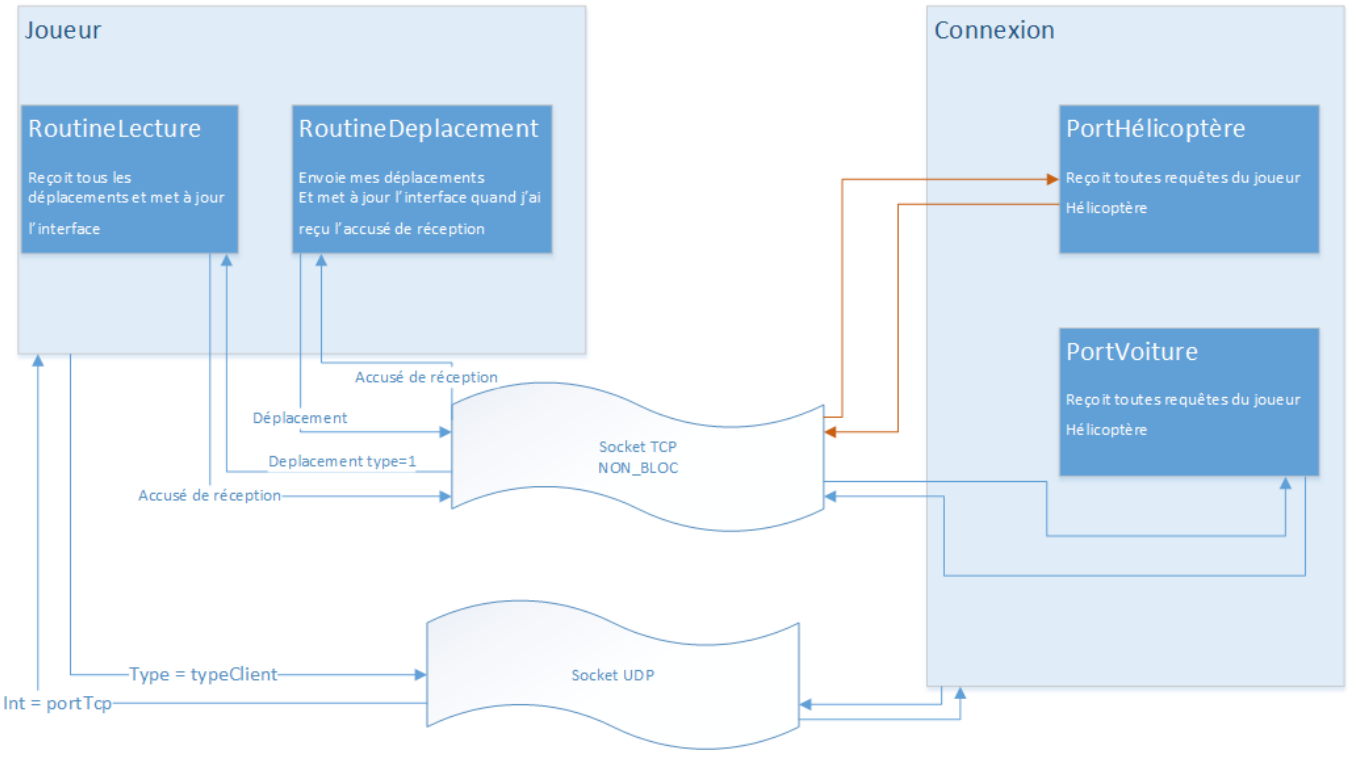
c. Reception des déplacements

Une fois l'initialisation terminée le client va créer un thread pour réceptionner les données de la socket. C'est dans ce thread que tous les messages de déplacements arriveront et que tous les messages d'accusé de réception partiront.

Nous avons fait en sort que la réception et l'envoi des messages soient répartis entre deux threads du client. La réception des messages est donc gérée par une routineLecture, alors que les envois de déplacements du joueur sont quant à eux gérer par la routineDeplacement.

Lors de la réception d'un message dans la socket tcp, il faut pouvoir identifier quelle type de données je reçois. Nous avons décidé d'identifier nos messages avec un champ type. Quand ce champ type est à 1, c'est que nous recevons un déplacement.
Ainsi nous savons que ce qui suit dans le message est une structure de Deplacement. Il sera donc lu et traité en conséquence.

Pour ce qu'il s'agit du traitement du Déplacement, nous y reviendrons plus en détails dans une partie à venir.



d. Gestion des figurants au niveau du Joueur

Au niveau du client, nous concrétisons la vie de la carte et des déplacements des Figurants avec les joueurs par l'affichage. Mais pour cela il faut pouvoir d'un côté récupérer les informations nécessaires à l'affichage et de l'autre savoir comment les traiter en conséquence. Or nous avons vu précédemment que nous avons choisit de n'envoyé que les déplacements qui changent dans la carte. Mais pour cela il faut donc pouvoir stocker ses informations sur le client pour qu'il suive rapidement ce qu'il se passe dans le jeu.

Pour cela nous avons décidé de stocker chaque Figurant dans une liste chainée. Cela va nous apporter de la mobilité au niveau de la façon de stocker les informations. L'autre façon aurait été de faire un tableau mais cela implique que la taille y a été imposée à la création. Ce qui n'ai dans notre cas pas possible puisque ces informations sont récupérée du serveur au démarrage.

Au démarrage tous les Figurants envoient aux joueurs leur deplacement, or au niveau du client, nous ne connaissons pas ce nouvel arrivant. C'est pourquoi nous l'ajoutons dans notre liste chainée. Comme cela si lors de son prochain déplacement, le Figurant aura deux positions potentielles. Il y a l'ancienne, celle qui est stocké en local dans notre liste chainée, et la toute nouvelle que l'on vient de recevoir. Nous serons alors capables de traiter les différences de façon précise et rapide. Nous reviendrons plus tard sur le déroulement de ce traitement.

Comme nous l'avons vu plus tôt chaque Figurant est identifié par un identifiant qui lui est propre, nous sommes donc tout à fait capable de le retrouver de dans notre liste grâce à une recherche de cet id.

e. Affichage des informations de jeu

La gestion des informations de jeu se fait lors de la réception des déplacements de figurant.

On compare avec la liste chainée l'état d'un figurant à son précédent déplacement à celui que l'on vient de recevoir, on peut donc en déduire l'état de celui-ci et l'afficher dans les informations :

Si précédemment c'était un piéton et que c'est devenu un terroriste au nouveau déplacement.
Alors on déduit la somme volée par le terroriste de la banque et on affiche le message suivant :

```
Un Cambrioleur (%s) est apparu [%d,%d] ! Vol de %dk euro ALERTE GENERAL !!! Banque : %d\n
```

Si précédemment il était sur une case prison et que maintenant il se déplace vers une autre case alors c'est qu'il a été libéré.
On affiche dans ce cas :

```
Un Terroriste (%s) a été remis en liberté [%d,%d] :(
```

Si c'était un terroriste et que son déplacement va sur une case prison alors on rajoute la somme volé à la banque du jeu et on affiche :

```
Un Terroriste (%s) a été mis en prison [%d,%d] on récupère %dk euro banque : %d :)
```

Si un terroriste de transforme en piéton et qu'il ne va pas en prison c'est qu'il c'est blanchie dans un repère de bandit on affiche donc :

```
Un Terroriste (%s) a été blanchi [%d,%d] :(
```

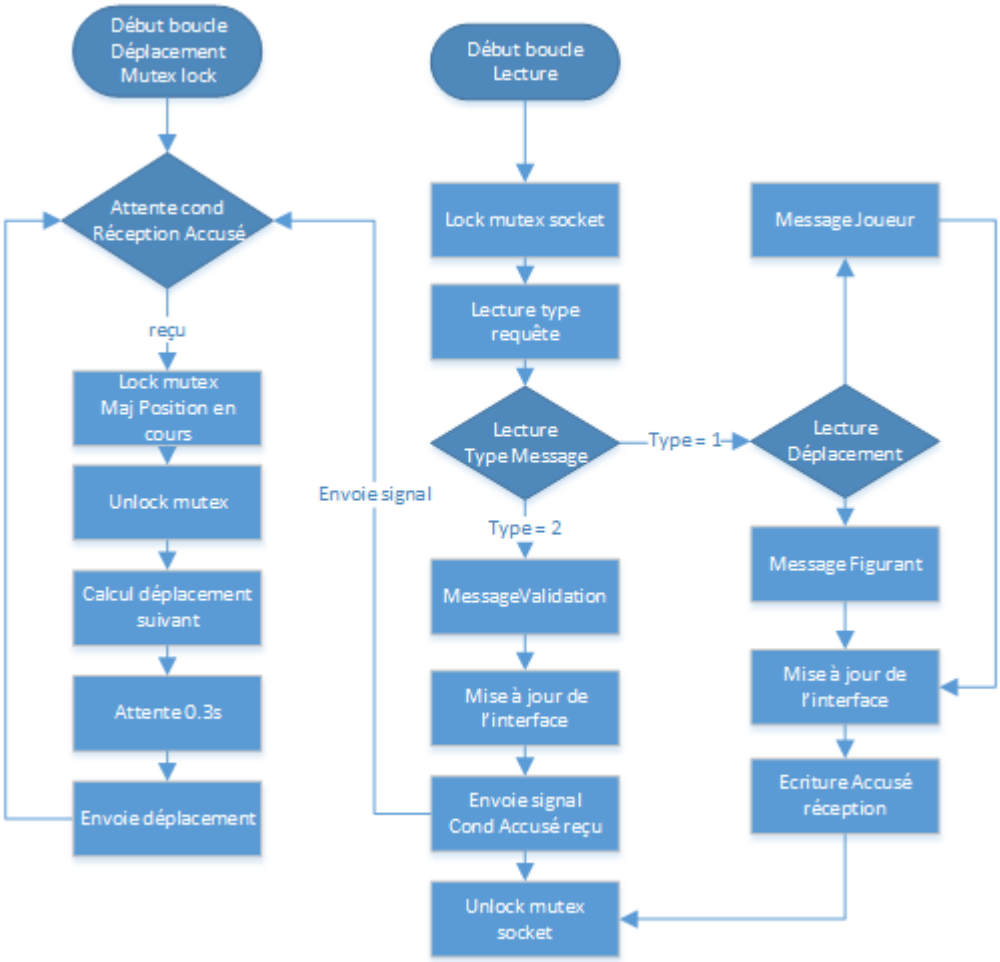
L'ensemble de l'affichage de ces informations se fait dans une section bien particulaire de notre interface graphique. Il s'agit de la section d'information.

f. Partage de la socket dans le Client

Dans l'application cliente, nous avons créé deux thread pour partager la lecture/écriture de la socket tcp en fonction des conditions. Nous avons donc un thread qui lit les déplacements que lui envoient les Figurants. Une fois qu'il les a reçu et traité (à savoir affiché et stocké dans la liste), il envoie dans la socket un accusé de réception. En parallèle, le thread de déplacement quant à lui écrit dans la socket pour envoyer son déplacement. Mais lui aussi attend un accusé de réception.
Nous avons donc un problème de coordination. En effet si le thread de déplacement attend son accusé de réception et que c'est le thread de lecture qui le lui lit ; il va dans un premier temps bloqué le thread de déplacement car il attendra son accusé, mais aussi le thread de lecture ne lira pas correctement les données qu'il recevra (avec risque d'erreur).

Pour éviter cela nous sommes dans l'obligation d'ordonner la répartition des tâches. Nous avons donc mis en place un mutex sur la socket. Ainsi la lecture se fera uniquement par le thread qui détiendra le mutex.
Au démarrage, c'est le thread Déplacement qui va prendre le mutex de la socket. Ensuite Il va entrer dans sa boucle principale. Cette boucle commence par la vérification que l'accusé de réception du déplacement a bien été reçu (ce signal ne peut qu'être envoyé par le thread de lecture). Une fois que la condition de réception d'un accusé de déplacement est vérifiée, le thread peut enfin continuer sa routine. Lorsqu'il va se remettre en route il pourra enfin libérer le mutex, puis le reprendre pour mettre à jour les données de déplacement. Une fois les données de déplacement il va pouvoir les envoyer sur la socket. Ensuite c'est au tour du thread de lecture de prendre le relai. Il va pouvoir lire le déplacement ou l'accusé de réception qu'il y aura dans la socket (choix grâce au type de message). Le thread de lecture va donc recevoir des messages tant que l'accusé de réception du déplacement ne sera pas reçu. Une fois qu'il sera reçu, le thread de déplacement va pouvoir être réveillé au moyen d'un signal sur la condition qu'il attendait (réception accusé de réception). Le mécanisme se répète jusqu'à ce qu'un ordre soit reçu.

Schéma de réceptions et d’émissions de déplacements

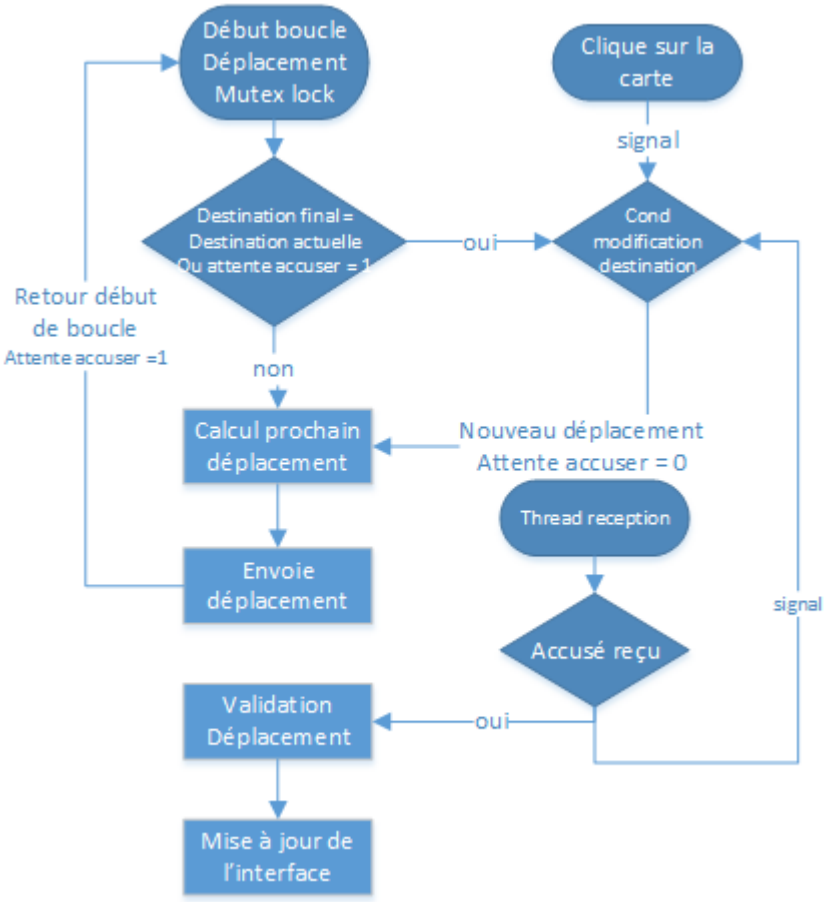


g. Déplacement du joueur

Le déplacement du joueur se fait par un thread qui se nomme routine Déplacement, se thread a pour objectif d'envoyer les demandes de déplacement au gestionnaire de connexion via la socket tcp.

Pour que le joueur se déplace librement nous avons mis en place quelques outils.
Tout d'abords, chaque routine, quand elle est créée, reçoit en paramètre une structure dans laquelle est renseignée l'ensemble des choses qu'elle a besoin. Pour commencer, chaque thread doit savoir ou on se trouve dans nos déplacement (notamment le thread de Déplacement et le « main » du Client), pour cela nous utilisons donc un pointeur vers une structure de position. Ensuite le thread a besoin de savoir comment se comporter par rapport à ses frères. Pour cela on utilise donc des pointeurs vers le mutex (pour bloquer l'accès à la socket) et un autre vers la variable de condition (accusé de réception reçu).

La structure de position nous permet de garantir que le mouvement que je souhaite faire sera fait. Car il se peut qu'il ne soit pas réalisé tout de suite car le thread dort (il attend son accusé de réception). Lorsque le joueur clique sur une nouvelle position alors il va réveiller le thread de Déplacement (envoie un signal avec la variable de condition accusé de réception).



h. Déplacement d’un Figurant

Dans le thread de lecture de la socket nouvellement créée nous allons nous trouver dans une boucle infinie.
Dans cette boucle infinie nous allons tenter de lire la socket, si on arrive pas à lire de données nous retentons quelques millisecondes plus tard.

Si on arrive à lire dans la socket nous lisons d’abord le type pour savoir le type de structure à suivre. 1 pour une structure déplacement ,2 pour un accusé de réception.

Si le type vaut 1 c’est que l’on reçoit un message soit d’un autre joueur soit d’un figurant qui se déplace, dans ce cas on lit la structure entière pour le déplacement (car on a le type avant).

Puis, première chose, si le champ joueur du déplacement est à 1, alors c’est l’autre joueur qui se déplace.
Lors du lancement du thread l’ancienne position du second joueur vaut -1 à chaque déplacement on efface l’ancienne position du joueur sur la carte, on met la nouvelle position à la place de l’ancienne position et on se place à la nouvelle position pour inscrire le joueur sur la carte, et enfin on actualise la carte dans notre fenêtre graphique ncurses.

Lorsque l’on efface l’ancienne case, on vérifie grâce à une recherche dans la liste chainée des figurant si il en existe un à l’ancien emplacement du joueur, si il y avait quelqu’un sur la même position que le joueur alors à la place d’effacer la case par du vide on affiche ce figurant.

Si le champ joueur n’est pas à 1 alors c’est un figurant.
La gestion des coordonnées de tous les figurant est faite par listes chainées, à chaque fois qu’on reçoit un déplacement d’un figurant, on vérifie si on a déjà reçu un déplacement de lui en recherchant son id dans la liste chainée, si il n’y est pas on l’ajoute , si il y est c’est qu’il est affiché sur la carte, il va donc falloir l’effacer de la même manière qu’on efface un joueur, ensuite on va donc ensuite actualiser les informations de coordonnées et type du figurant dans la liste chainée avec les informations reçu , enfin on affiche à la nouvelle position le figurant et on actualise l’interface graphique ncurses.

6. Ce qu’il faut retenir

Le but de ce projet était de réaliser une application réparti et en réseau dans un langage système bas niveau. Dans ce projet nous avons dû utiliser les outils système tels que les outils ipc ainsi que l’utilisation de la bibliothèque pthread.

L’utilisation de ces ressources nous a permis de faire une application répartie pour simuler l’ensemble des mouvements des figurants. Mais aussi de pouvoir créer une application dans laquelle nous contrôlons l’ensemble des actions que l’on fait sur des données qui devaient être partagée entre plusieurs processus.

Le contrôle et l’ordre d’applications des différentes actions nous a permis de faire en sorte que notre application ne génère des erreurs sans arrêts.

De plus le traitement systématique des différents cas possible lors de n’importe quel appel système nous permet aussi de garantir son bon fonctionnement. A noter que toutes les erreurs que nous traitons ne sont pas nécessairement de vrai erreurs qui vont provoquer l’arrêt de notre application.